**CS-551**

**Project 2**

**Design document and Manual**

**By –**

**Goutham Kannan (CWID- A20361163)**

**And**

**Imran Ali Usmani (CWID- A20362180)**

**Flow control diagram for deposit(), retrieve() and clear()**

1. **Deposit()**

User Level

Deposit(message,recivers)

Kernel Layer

PM SERVER

PM\_DEPOSIT

Do\_deposit()

Do\_deposit()

**Flow control for do\_deposit(Message , Recievers)**

Check if mailbox is full..?

**Yes**

Report Error :Mailbox full

**No**

Move to the next available messagebox

Copy the message data to the mailbox(sys\_datacopy)

Did the message get copied?

Report Error

**Yes**

1. **Retrieve()**

User Level

retrieve (receiver name)

Message Content

Kernel Layer

PM SERVER

PM\_RETRIVE

Do\_retreive()

Do\_retreive()

**Flow control for retrieve (receiver\_name)**

Check if mailbox is not empty

**No**

Report Error :Mailbox empty

**Yes**

Check if mailbox has any message for the receiver

**No**

Report Error: No message available

**Yes**

Copy the message data from the mailbox(sys\_datacopy) to user process

**Flow control for clear()**

Clear all the existing messages in the mailbox and their receiver entries

**Blocking or Non-Blocking and Reasons**

Both the system calls deposit() and retrieve() have designed to be **non-blocking**. This would give the user more flexibility in terms of implementation. It can use it as blocking or non-blocking at its end. If it wants it to be non-blocking, it will call it only a finite number of times and move ahead with the rest of the work. But if its wants to be blocking and does not want to move ahead without depositing or retrieving successfully, it just have put the system call in an infinite loop like:

*while( ret<1 ){*

*ret = deposit();*

*}*

**User Manual:**

## 1. Name

deposit() – deposit a message into mailbox

**Synopsis**

**#include <[mailbox.h](http://linux.die.net/include/unistd.h)>**

**int deposit(char** \* *string***, unsigned int** *receiver*);

**Description**

The deposit() function shall attempt to write/submit a message(**char** \* *string*) into the mailbox, for receiver no.(the position of the bits which are set 1 in **unsigned int** *receiver*).

The maximum length of the *string* is allowed upto 49 characters. A single message can be deposited for multiple number of receivers. Maximum number of receivers can the equal to the size of **int** in bits, i.e. if **int** is of size 4 bytes (32 bits), a single message could be sent to receiver ids numbering from 0 to 31. It internally uses kernel call sys\_datacopy() to copy the message from calling user process to the pm server’s mailbox.

If the message is to be deposited for receiver with id 1 than set first bit of *receiver*

**unsigned int** *receiver=1;*

Similarly, if the message is to be deposited for receivers with id 1 and 3 than set first and third bit of *receiver(101 in binary) ,* so

**unsigned int** *receiver=5;*

**Return Value**

Upon successful completion, returns the number of bytes copied. Otherwise, returns (**0**) if the mailbox is full. And returns (**-1**) if fails to write/deposit a message.

**Exceptions**

1. If it is unable to write data to the mailbox, i.e. if sys\_datacopy() fails, don’t keep waiting and return the control back to the user process, with return value(-1). This makes the system call non-blocking. Now it is up to the user to retry depositing the message to the mailbox or to move ahead.

2. If the mailbox is full, it shall not wait for the mailbox to become empty. It will return back to the user process, with the return value(0). This again makes the system call non-blocking. It is up to the user to retry depositing the message after the space is available in the mailbox or it should move ahead.

**Source**

minix/servers/pm/misc.c

## 2. Name

retrieve () – deposit a message into mailbox

**Synopsis**

**#include <[mailbox.h](http://linux.die.net/include/unistd.h)>**

**int retrieve(char** \* *string***, unsigned int** *receiver*);

**Description**

The retrieve() function shall attempt to extract a message which have receiver id as **unsigned int** *receiver*, from the mailbox and copy it to the address **char** \* *string*. The message could be extracted for only one receiver at a time. If the receiver id is **n**, the **n**th bit of the **unsigned int** *receiver* is set to one.

The maximum length of the *string* to be extracted could be upto 49 characters. A single message can be deposited for multiple number of receivers. Highest receiver id can the equal to the size of **int** in bits, i.e. if **int** is of size 4 bytes (32 bits), a single message could be sent to receiver ids numbering from 0 to 31. It internally uses kernel call sys\_datacopy() to copy the message from the pm server’s mailbox to calling user process address space.

If the message is to be retrieved for receiver with id 1 than set first bit of *receiver*

**unsigned int** *receiver=1;*

similarly, if the message is to be retrieved for receivers with id 5 than set fifth bit of *receiver(10000 in binary) ,* so

**unsigned int** *receiver=32;*

**Return Value**

Upon successful completion, returns the number of bytes copied. Otherwise, returns (**0**) if no message exists for that receiver id in the mailbox. And returns (**-1**) if it is unable to retrieve, i.e. unable to copy data from mailbox.

**Exceptions**

1. If it is unable to write data from the mailbox to the calling user process’s address space, i.e. if sys\_datacopy() fails, don’t keep waiting and return the control back to the user process, with return value(-1). This makes the system call non-blocking. Now it is up to the user to retry retrieving the message from the mailbox or to move ahead.

2. If no message exists in the mailbox for that receiver, it shall not wait till the message arrives for that receiver in the mailbox. It will return back to the user process, with the return value(0). This again makes the system call non-blocking. It is up to the user to retry retrieving the message till the message arrives in the mailbox or it should move ahead.

**Source**

minix/servers/pm/misc.c

## 3. Name

clear() – deleting all the messages of the mailbox

**Synopsis**

**#include <[mailbox.h](http://linux.die.net/include/unistd.h)>**

**int clear(**);

**Description**

The clear() function shall delete all the messages present in the mailbox. It also clears all the receiver ids present in the mailbox for each message.

**Return Value**

Upon successful completion, (1). Otherwise, returns (**-1**) if it is unable to clear the mailbox.

**Exceptions**

1. If it is unable to clear data of the mailbox, it doesn’t keep waiting and return the control back to the user process, with return value(-1). This makes the system call non-blocking. Now it is up to the user to retry clearing the mailbox or to move ahead.

**Source**

minix/servers/pm/misc.c

## 4. Name

setup() –setting up the mailbox

**Synopsis**

**#include <[mailbox.h](http://linux.die.net/include/unistd.h)>**

**int clear(**);

**Description**

The setup() function shall setup a fresh MailBox with now values inside it.

**Return Value**

Upon successful completion, (1). Otherwise, returns (**-1**) if it is unable to setup the mailbox.

**Deadlock conditions and Handling**

Since both of the system calls deposit() and retrieve() are non-blocking, this removes the condition of hold and wait to happen even if multiple or concurrent deposit() and retrieve() calls are made. So our calls would not allow deadlock to happen.

**Live-lock (detection and recovery)**

But there is a possibility of live-lock to occur if the user calls the system call in an infinite loop in the user process and the mailbox is full(in case of deposit()) or if no message exists(in case of retrieve()) for a long time. In this case if the user process is in live lock for too long, the user can search for the return value (-2) by the system call. If it receives return value (-2), it can get out of the live-lock by terminating the infinite loop or even clearing the mailbox or doing a deposit() in case of retrieve() live-lock to make the retrieve() successful.

**Exceptions and Handling**:

**1.** If the mailbox is full and user tries to deposit a message. The system call does not wait for the mailbox to be empty. It returns the control back to the user with the return value (0). This removes the condition of busy waiting, and hence the deadlock.

**2.** If the user tries to retrieve a message but no message exists. The system call does not wait for the message to arrive. It returns as soon as it does not find a message, with the return value (0). This removes the condition of busy waiting, and hence the deadlock.

**3.** If the mailbox is empty, but the system call is unable to copy message from the calling user process’s address space to the pm server’s mailbox. The system call does not retry copying, it just returns control to the user with return value (-1) implying that sys\_datacopy() failed. Now it is up to the user to retry after sometime or to move ahead. This removes the condition of busy waiting, and hence the deadlock.

**4.** If the system call finds a message to retrieve, but the system call is unable to copy message from the mailbox to the calling user process’s address space. The system call does not retry copying, it just returns control to the user with return value (-1) implying that sys\_datacopy() failed. Now it is up to the user to retry after sometime or to move ahead. This removes the condition of busy waiting, and hence the deadlock.

**How message ordering is maintained**?

The mailbox contains 16 message boxes to deposit a message. To maintain the message ordering, i.e. a message deposited first should be retrieved first, we have implemented the mailbox as similar to a queue, i.e. first-in-first-out. So whenever a new message is deposited, it is deposited at the back of the queue. When a message in deleted all the messages behind it are moved 1 box to the front. And for retrieving a message, the search is started form the front and whenever a message is found with the required receiver id, it is retrieved. So this maintains the ordering such that a message (for a particular receiver) if arrives first, is retrieved first.

*Deposit Message1 for Receiver 1*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| M1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

*Deposit Message2 for Receiver 1*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| M1 | M2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

*Retrieve – Receiver 1*

M1 – will be retrieved not M2 as the mailbox follows FIFO

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| M2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

*Deposit Message 3 for Receiver 1*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| M2 | M3 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

*Deposit Message 4 for Receiver 1*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| M2 | M3 | M4 |  |  |  |  |  |  |  |  |  |  |  |  |  |

*Retrieve – Receiver 1*

Message M2 will be retrieved ahead of M3 and M4 as M2 arrived the earliest.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| M3 | M4 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

**Test case scenarios**

**1. 1 deposit and 1 retrieve:** Do a single deposit and retrieve and see if the message is retrieved successfully.

**2. Multiple deposits for multiple receivers and 1 retrieve concurrently:** Do multiple deposits for multiple receivers and retrieve a message for a single receiver simultaneously and see if the message is retrieved successfully.

**3. 1 deposit and multiple retrieves concurrently:** Do a single deposit and retrieve a message for a multiple receivers with different receiver id and see if the message is retrieved only for the receiver for which the retrieve call was made.

**4. Multiple deposits for multiple receivers and multiple retrieves concurrently:** Do multiple deposits for multiple receivers and do multiple retrieves for different receiver ids and see if all the message is retrieved successfully.

**5. Ordering check:** Do multiple deposits for a particular receiver. Also do some deposits for other receivers in between. Retrieve the messages of that particular receiver and see if the message is retrieved in the same order as they were submitted.

**6. Endurance test:** In one process do multiple deposits in an infinite loop. In another process do multiple retrieves in an infinite loop. Run these two processes concurrently, see if no error occurs.